YADAS/YATRAN Tutorial, Version 2008.2

Todd L. Graves

February 6, 2008

1 Introduction

This document is an introduction to computing with Markov chain Monte Carlo using YADAS and its interface YATRAN. It is likely that neither of these two ways of accessing YADAS functionality is as straightforward as you hope: YADAS itself requires you to write or modify Java code with characteristics unique to YADAS, while YATRAN will look much friendlier but is early in development and therefore supports only a small subset of YADAS. Still, we very much hope that you will persevere and become a YADAS user and we are willing to help you do so. This document works through the first example on the YADAS web site basically line-by-line; future installments of this tutorial are likely to contain reliability examples or be tailored to other interests of yours.

For more information, one can go to the YADAS website `yadas.lanl.gov`, which contains eight more examples, and especially to the downloads page `http://www.stat.lanl.gov/yadas/node1.html#download`, and also to Todd's papers web page `http://www.ccs.lanl.gov/ccs6/staff/TGraves/pdfs.html`. Another source of information is on `bubs`, in `projects/yodas/tutorial08`. Of course, you can also go to Todd, Richard, or Mike for help.

2 Setting Up YADAS and running an example

The first step in installing YADAS on your system is to acquire a copy of the file `yadas.jar` and saving it somewhere. (It can be saved under a different name if you choose.) A `.jar` file is a "Java archive", which contains compiled versions (*classes*) of Java code that you can use. Next, you need to tell your system that one of the places it should look for Java class files is in this `.jar` file. This is done by setting an environment variable called `CLASSPATH`. On a Windows system, go to the Control Panel, Performance and Maintenance, then to System, then to Advanced, and there will be a button that allows you to set environment variables; probably you will have to define a new one called `CLASSPATH`. Assuming that you have a file called `yadas.jar` in the folder `c:\\Java`, a suggested value of the `CLASSPATH` variable is `.;c:\\Java\\yadas.jar`. (If you already have a `CLASSPATH` variable defined, append the jar file and, if necessary, the current directory to the existing value of this variable.) This means that whenever you compile (using `javac`) or run (using `java`) a Java class, the compiler or interpreter will look for class files in the current directory (`.`) and then in the `yadas.jar` file. On a Macintosh, open a terminal window and execute the command `export CLASSPATH=''.:/Users/tgraves/Java/yadas.jar''`, with the path changed to where you put the `yadas.jar` file on your own system. This statement should be placed in your `.profile` file so that you don't have to execute it each time you open a new terminal window, and the jar file should be appended to any existing value of the `CLASSPATH` variable. I don't know how to do these things on Linux but it must be similar to the Mac process.

It is likely that these instructions will be wrong or at least difficult to follow; if so, do not hesitate to get help from Todd or Mike.

Next, convince yourself that Java is installed on your system and ready to be used. On a Mac, this is already taken care of. On a Windows machine, I think you need to go to `java.sun.com`, then to "Java SE" from the Popular Downloads menu on the right, then hit the Download button under JDK 6 Update 4 or whatever is current (you need the JDK, "Java Developer's Kit"). Install that somewhere and locate

the resulting `javac.exe` file (it's probably somewhere like `c:\\Program Files\\Java\\jdk1.6.0_04\\bin`), and ensure that the folder it's stored in is in your `PATH` environment variable, using the same process as you used to set your `CLASSPATH`.

Next, acquire the `Examples.zip` file, which was probably mailed to you as something called `Examples.zzz` since our mail server strips off zip files; rename it and unzip it. Go to `Examples/1` in a terminal, DOS, or CYGWIN window, and execute the command `javac OneWayAnova.java` (you're compiling the OneWayAnova class). You should have generated a new file called `OneWayAnova.class`. Now run the command `java OneWayAnova 10000`. This runs the MCMC algorithm defined in the OneWayAnova.java file for 10000 iterations, and generates output files `mu.out, theta.out, sigma.out, and delta.out`, which you can read into R and use to create summary statistics and plots. There's not much point in doing that yet, since you don't even know what the model is. We'll tell you that, but first let's make sure YATRAN is also set up and ready to run. Another good idea at this point is to unzip the `yadas-src.zzz` file so that you can look at the YADAS source code when you want to. Someday you may want to compile this source code and use the resulting compiled classes instead of the `.jar` file, but we won't worry about that yet.

3 Setting Up YATRAN

The instructions are in the "YATRAN User Guide", by Richard Klamann.

4 The First Example: One Way Anova

The first example model is a one-way analysis of variance. We have observed Gaussian data

$$y_i, 0 \leq i < n_y.$$

These data are stratified into different groups, with the ith data point belonging to group g_i, where $g_i \in \{0, 1, \ldots, n_g - 1\}$. We allow each group to have a different mean, so that the data distribution is

$$y_i \sim N(\mu_{g_i}, \sigma^2) \ (0 \leq i < n_y).$$

We assume a hierarchical model for the group means:

$$\mu_j \sim N(\theta, \delta^2) \ (0 \leq j < n_g).$$

The standard deviation parameters are given gamma prior distributions with known hyperparameters, i.e.

$$\sigma \sim \Gamma(a_\sigma, b_\sigma) \text{ and } \delta \sim \Gamma(a_\delta, b_\delta),$$

where we use the shape-scale parameterization so that $E(\sigma) = a_\sigma b_\sigma$. We use an improper flat prior for θ ("uniform on $(-\infty, \infty)$"). We wish to sample from the joint posterior distribution of $(\mu_0, \ldots, \mu_{n_g-1}, \theta, \sigma, \delta)$.

5 The YATRAN Code for One Way Anova

This is discussed beautifully in the "YATRAN User Guide," by Richard Klamann, on pages 2-5.

6 The YADAS Code for One Way Anova

In this section we go over the code in `OneWayAnova.java`, more or less line-by-line, trying to explain all the concepts that arise. The format is that a piece of the code will be reproduced in `typewriter font`, and below that piece of code we discuss it. This probably appears to be quite a complicated example. Unfortunately YADAS is not currently set up to do easy things easily. The most promising method of creating a new application is to start with an application already written by yourself or someone else and make changes to

it so it can do what you want. Here we will try to tell you what every line in this file is doing and introduce all the relevant concepts we can think of with this example.

First, what is this file? It's the file that defines the class `OneWayAnova`, as can be seen from both the line 13

```
public class OneWayAnova {
```

and from the fact that the file is called `OneWayAnova.java`; both of these things are mandatory if you want to create an executable called `OneWayAnova.class`. A class definition file defines how Java can create an object belonging to that class, and furthermore what can be done with and to such an object after it has been created. A class consists of data and *methods*. Methods are functions. Most methods can be called using syntax like the following: suppose `theta` is an object of class `MCMCParameter`, which is one of the most famous classes in YADAS. `MCMCParameter`s have a method called `update`. If you want to "update theta" i.e. call the `update` method of the object `theta`, the code that does this is `theta.update()`. (You will probably never have to write this exact piece of code in any application.) Another method of the class `MCMCParameter` is `getValue`, which allows you to query the current value of the kth element of `theta` by `theta.getValue(k)`, i.e. if θ is a vector of at least 9 elements, you can get the current value of θ_8 with `theta.getValue(8)`, and θ_0 by `theta.getValue(0)`. Indexing in Java starts at 0 like in C, not at 1 like in R! A special method that a class may or may not have is its `main` method, which starts at line 16 with

```
    public static void main (String[] args) {
```

If a class has a main method, you can run it "as an application" from the command line e.g. with `java OneWayAnova`.

```
/*
   OneWayAnova.java:  Example 1 on the YADAS tutorial.
   Data y_{ij} ~ N ( \mu_{i}, \sigma^2 ),
   \mu_{i} ~ N ( \theta, \delta^2 ),
   \theta ~ flat prior on (-\infty, \infty),
   \sigma ~ Gamma (a_\sigma, b_\sigma),
   \delta ~ Gamma (a_\delta, b_\delta).
*/
```

Everything between $/*$ and $*/$ is a comment, and everything on a line beginning // is also a comment.

```
import java.util.*;
import gov.lanl.yadas.*;
```

The first coding lines in a class file, if necessary, will be `import` statements that list the *packages* that the class relies on: here, the `java.util` and `gov.lanl.yadas` packages. If these lines are omitted, the compiler won't know some of the things it needs to know to compile this class. For example, if the yadas package isn't imported, the compiler won't know what to do with `DataFrame`, `MCMCParameter`, and loads of other things. The `java.util` package is needed for `ArrayList`, if I remember correctly.

```
public class OneWayAnova {
    public static void main (String[] args) {
```

We've mentioned these lines already; here we begin the definition of the `OneWayAnova` class and its `main` method, which is what is executed when the class is run from the command line. `public`, `static`, and `void` are keywords that describe properties of the main method; `main` methods are always defined to be `public static void`. You probably don't need to understand these keywords now, but here's an introduction if you're curious: `public` defines from where this method can be called (other possibilities would include only from this class, only from classes in this package, etc.); `static` is a somewhat advanced concept that basically means that the compiler needs only one definition of this method, even if you have several different

`OneWayAnova` objects running around; and `void` means that this method does not return anything, although it may create side effects such as output files. Note that the `main` method is written so that in principle it can accept an array of Strings as arguments: this is also required for a `main` method. (In Java, an *array* is much like what is called a vector in R: an ordered list of things of the same type. The term array is used for one-dimensional arrays as well as higher dimensional arrays. Arrays, when initialized, are defined to have a certain length, and their lengths can not change later. If you need to keep objects together in lists whose lengths may change, use a Java *Collection*, one example of which is `ArrayList` mentioned above.)

```
int B = 1000;
String direc = "";
String filename = "Ex1data.dat";
String filename2 = "Ex1mu.dat";
String shortfilename = "Ex1scalars.dat";
```

Here we are starting to *declare* and *initialize* some variables. Each of these lines does two things: allocate storage space for a variable, and then give it an initial value. Instead of the first line, we could have written

```
int B;
B = 1000;
```

which divides up the two tasks into two statements. Note that a statement in Java must end with a semicolon. What we are actually trying to accomplish with these five lines is defining default values of various things that we can override if we want to with command line arguments. Here `B` is the number of MCMC iterations, and the Strings define the locations of input files: we might want to run this code on many different data sets.

```
try {
    if (args.length > 0)
        B = Integer.parseInt(args[0]);
    if (args.length > 1)
        direc = args[1];
    if (args.length > 2)
        filename = args[2];
    if (args.length > 3)
        filename2 = args[3];
    if (args.length > 4)
        shortfilename = args[4];
}
catch (NumberFormatException e) {
    System.out.println("Poor argument list!" + e);
}
```

Here we're completing the task of processing command-line arguments, so that we can use this code to run more or less than 1000 iterations or analyze data in different files and directories than the defaults. Here we learn a few things: how to write simple `if` statements, how to ascertain how many objects are in an array, how to refer to an element in an array (again recall that array indices start with zero!), and how to send a message to standard output. Advanced topics here are coercing a String into an integer using `Integer.parseInt`, and exception handling using `try` and `catch`. Basically someone could have used a command line argument for `B` which can't be interpreted as an integer, and Java wants to allow the possibility of the programmer handling this gracefully, so it "throws an exception" in the code encircled by `try`, which can then be handled by the `catch` code. I haven't yet figured out how to take good advantage of this in YADAS.

```
DataFrame d = new DataFrame (direc + filename);
DataFrame d2 = new DataFrame (direc + filename2);
ScalarFrame d0 = new ScalarFrame (direc + shortfilename);
```

Here for the first time we are introduced to classes specific to YADAS, namely `DataFrame` and `ScalarFrame`, whose purposes are to read mixed integer and real input data from files and then allow you to refer to the data by the names of the variables, like data frames in S and R. Open `data.dat` from the `Examples/1` folder to see what files that can be read by `DataFrame` look like. The first line is the number of lines of data. The second line is a list of variable names, separated by pipes (|). (You don't have to use pipes, but it's a little harder to use some other delimiter.) The third line is a list of variable types: `r` for real, `i` for integer, and `s` for String, although I very rarely use strings. There are a few surprises about what sorts of variables need to be real rather than integer: for example, binomial sample sizes and numbers of successes are real. The fourth lines and below contain the data, still separated by pipes. All data frames are rectangular (i.e. all variables have the same length.) Now that you've read the data into a `DataFrame` and called that `d`, you can access the data y using `d.r("y")`, and the integer group labels using `d.i("group")`. `scalars.dat` is a sample YADAS scalar frame, which is like a data frame with all variables having length one. The initial value of θ, stored in the data frame `d0`, can be accessed using `d0.r("theta")`. Note that you don't have to explicitly call out whether a scalar frame element is real or integer, and it can't be a string.

Note that in Java, strings are concatenated using `+` as in `direc + filename`.

```
MCMCParameter mu, theta, sigma, delta;

MCMCParameter[] paramarray = new MCMCParameter[]
{
    mu = new MCMCParameter (d2.r("mu"), d2.r("mumss"),
                    direc + "mu"),
    theta = new MCMCParameter (d0.r("theta"), d0.r("thetamss"),
                    direc + "theta"),
    sigma = new MCMCParameter (d0.r("sigma"), d0.r("sigmamss"),
                    direc + "sigma"),
    delta = new MCMCParameter (d0.r("delta"), d0.r("deltamss"),
                    direc + "delta"),
};
```

This is where we announce what parameters we seek posterior samples of. We have previously learned how to declare and initialize an object; here we declare and initialize an *array*. This array consists of objects belonging to the YADAS class `MCMCParameter`. Note that we give names (`mu, theta,` etc.) to the elements of this array and declare single `MCMCParameter`s of these names in the first line in the above excerpt.

The initial definition of an `MCMCParameter` is an array of initial values, an array of step sizes, and a filename to which to send output for that parameter. For example, the initial values and step sizes for `mu` are in the `DataFrame d`, while the other parameters are of length one and initialized with values in the `ScalarFrame d0`. Output is sent to the same directory as contained the input files, and the filenames are `mu.out`, `theta.out`, etc.

```
MCMCBond databond, muprior, sigmaprior, deltaprior;

ArrayList bondlist = new ArrayList ();

bondlist.add ( databond = new BasicMCMCBond
               ( new MCMCParameter[] { mu, sigma },
                 new ArgumentMaker[]
                     { new ConstantArgument (d.r("y")),
                       new GroupArgument (0, d.i("group")),
                       new GroupArgument (1, d.i(0)) },
                 new Gaussian () ));
```

OK, the hard stuff is beginning. This is the start of the definition of the model. First we declare four objects that implement the YADAS interface `MCMCBond`, then we declare an `ArrayList` to hold them. `ArrayList` is a Java class that holds an arbitrary number of objects of arbitrary type(s) in a specified order; this `ArrayList` is not actually used for any critical purpose. You add elements to an array list using the method `add()`.

At this point it is not important to understand exactly what `MCMCBond` means in the design of YADAS. If you want to, probably the paper "Design Ideas for Markov Chain Monte Carlo Software," published in JCGS and available on my papers website

`http://www.ccs.lanl.gov/ccs6/staff/TGraves/pdfs.html`

is the best place to look. For the moment, a term in the unnormalized posterior distribution is defined using an `MCMCBond` (the unnormalized posterior is the product of the bonds). The word "bond" comes from imagining a chemistry interpretation of a graphical depiction of the model in which parameters are atoms, and more than two parameters can be bonded together with a single bond. What's a "term"? If you have several normally distributed data points, most likely all of their distributions will be defined together using a single `MCMCBond`.

It is important that you understand why the code means

$$y_i \sim N(\mu_{g_i}, \sigma^2) \text{ for } i = 0, 1, \ldots, n_y. \tag{1}$$

You should eventually be able to mimic this sort of code to define your own models. What have we done here? We have given the bond a name, "databond", which is not strictly necessary; it can be used for debugging. Then we have used the `BasicMCMCBond` syntax to define this part of the model. `BasicMCMCBond`s think of pieces of statistical models in the following way. We begin with some parameters:

```
new MCMCParameter[] { mu, sigma }
```

is the syntax for defining a new array of objects belonging to the `MCMCParameter` class, and containing the two existing `MCMCParameter`s `mu` and `sigma`. Now we want to process these parameters together with some data and constants to generate vectors that can be run through standard probability density functions. In this example, we have to process μ and σ so that we can use the standard `Gaussian` function, which knows how to calculate

$$\text{Gaussian}(d, m, s) = -\log(s) - \frac{1}{2}\log(2\pi) - \frac{1}{2s^2}(d-m)^2, \tag{2}$$

the log of the standard univariate Gaussian density function as a function of its data d, its mean m and its standard deviation s. To use this in a specific problem, we need to build an array of d's, an array of m's, and an array of s's, and we do this using YADAS objects called `ArgumentMaker`s. First, we use a `ConstantArgument` to indicate that the array of data y in the `DataFrame d` are to play the role of the Gaussian data; a `ConstantArgument` is used when the contents of an argument are not going to change at any point of the algorithm. Then we use `GroupArgument`s to define the m and s arrays: first, the line

```
new GroupArgument (0, d.i("group"))
```

indicates that the parameter `mu` will be used to build the mean. It uses a 0 to point to the zeroth (i.e. first) element of the array of parameters defined earlier, and this zeroth element is the parameter `mu`. Next it *expands* this parameter so that it has the appropriate length (i.e. the same length as the data vector), and so that the correct μ_j correspond to the data in the jth group. It does this using the array of integers in the `data.dat` input file and stored in the `DataFrame d` under the name "group". This works in the same way as it would in R if you had a vector `mu = c(mu0, mu1, mu2)` and a vector `g = c(rep(0,10), rep(1,10), rep(2,10))` and let `m = mu[g+1]`. (Once again, the +1 is necessary because array indices start with 1 in one language and 0 in the other.) The definition of the array of standard deviations is similar: the 1 points to `sigma` and `d.i(0)` is an array of all zeros of the length of the variables in the `DataFrame d` (i.e. one zero for each data point).

If you hate parameterizing the Gaussian with its standard deviation, Hamada has written a GaussianVar, for example, and it's no trouble to write a `GaussianPrecision` for that matter.

```
bondlist.add ( muprior = new BasicMCMCBond
               ( new MCMCParameter[] { mu, theta, delta },
                 new ArgumentMaker[]
                     { new IdentityArgument (0),
```

```
                        new GroupArgument (1, d2.i(0)),
                        new GroupArgument (2, d2.i(0)) },
                    new Gaussian () ));

bondlist.add ( sigmaprior = new BasicMCMCBond
                ( new MCMCParameter[] { sigma },
                  new ArgumentMaker[]
                      { new IdentityArgument (0),
                        new ConstantArgument (d0.r("asigma")),
                        new ConstantArgument (d0.r("bsigma")) },
                  new Gamma () ));

bondlist.add ( deltaprior = new BasicMCMCBond
                ( new MCMCParameter[] { delta },
                  new ArgumentMaker[]
                      { new IdentityArgument (0),
                        new ConstantArgument (d0.r("adelta")),
                        new ConstantArgument (d0.r("bdelta")) },
                  new Gamma () ));
```

Here we define the remainder of the unnormalized posterior distribution. First, we define the prior distribution for the μs, which introduces the `IdentityArgument`: this makes a copy of one of the parameters without expanding or otherwise altering it. All the μs have the same mean and standard deviation: the scalar parameters `theta` and `delta`, respectively, so that these parameters have to be expanded to be the length of the data frame `d2`. `sigma` and `delta` are both given Gamma priors (in YADAS, the gamma distribution is parameterized by its shape and scale parameters). The various prior parameters are not estimated and are defined in the `ScalarFrame`. Note that no prior distribution is specified for `theta`; that means it is uniform on the entire real line (its prior is proportional to 1 everywhere).

```
MCMCUpdate[] updatearray = new MCMCUpdate[]
{
    mu, theta, sigma, delta,
};
```

This is the definition of the YADAS algorithm, in its simplest form. We build an array of update steps, and one iteration in the MCMC algorithm consists of looping through the elements of this array. Here we have defined an algorithm simply by listing all the parameters we are trying to estimate: this means that each of the elements of each of the parameters will be updated using variable-at-a-time Metropolis with a Gaussian proposal.

Since this part of the code is pretty simple to write, this is a good place to tell you about the "interface" in Java. An interface is a set of abstract descriptions of methods. Then a class "implements" the interface if it contains definitions for all the methods in the interface. This pays off in code like the above, where we make an array of objects that can in principle be quite different except that they all implement the same interface (in this case, `MCMCUpdate`), and then we can process them in a loop in ways permitted by the interface. Later in this YADAS code, we loop over the objects in this `updatearray` and call the `update` method of each, and one cycle through this loop is one iteration in the MCMC algorithm. In the example in this YADAS application, all the updates are parameters, and parameters have `update` methods that perform variable-at-a-time random walk Metropolis with Gaussian proposals. Different YADAS algorithms can be built by putting different things such as `MultipleParameterUpdate`s in this array.

There are people better than me at explaining object-oriented concepts. I like the treatment in Bruce Eckel's *Thinking in Java*, a considerable amount of which is available online at `http://mindview.net/Books/TIJ4`. Go there, click on "Sample: Front Matter + First 7 Chapters + Index", and read pages 15-28 if you like. Presumably `http://java.sun.com/docs/books/tutorial/` has some useful information as well, but I haven't read it.

One easy and useful way to modify the standard YADAS algorithm is to add step size tuning. This runs an experiment with different step sizes, then runs a logistic regression on number of accepted moves as a

function of log(step size) and chooses step sizes that should yield desired acceptance rates of about $1/e$; see my paper "Automatic Step Size Selection in Random Walk Metropolis Algorithms" (although this paper may have sections that end in midsentence). If we wanted to use this step size tuner for all the parameters in the one-way anova example, we would change the definition of `updatearray` to:

```
MCMCUpdate[] updatearray = new MCMCUpdate[]
{
    new UpdateTuner (mu),
    new UpdateTuner (theta),
    new UpdateTuner (sigma),
    new UpdateTuner (delta),
};
```

(That's the end of the new definition of `updatearray`. Now we move on to another piece of the code.)

```
for (int b = 0; b < B; b++) {
    if ((b/1000.0 - (int)(b/1000))== 0) System.out.println(b);
    for (int i = 0; i < updatearray.length; i++) {
        updatearray[i].update ();
    }
    for (int i = 0; i < paramarray.length; i++) {
        paramarray[i].output ();
    }
}
```

At this point we're wrapping up and discussing code that you probably won't have to change. This is the code that runs the algorithm and generates output. `B` is the number of MCMC iterations; after every 1000 we send a simple progress report to standard output. Every iteration consists of a call to the `update` method for each update in the algorithm, and at the end of an iteration we send the current value of each parameter to a file via the `output` method.

```
String acc;
for (int iii = 0; iii < updatearray.length; iii++) {
    acc = updatearray[iii].accepted();
    System.out.println("Update " + iii + ": " + acc);
}
for (int i = 0; i < paramarray.length; i++) {
    paramarray[i].finish();
}
```

In the last piece of code we send information about the acceptance rates of the various update steps to standard output, and then clean up by ensuring that all MCMC samples are actually written to the file rather than sitting in a buffer. That's it!

Exercises

When modifying a class, give the file and the class a new name (recall that this means changing the `public class OneWayAnova` line to, for example, `public class YADASTutorialExercise1` and saving the file as `YADASTutorialExercise1.java`) and recompile.

1. If you haven't already, run the `OneWayAnova` application, read the output into R, and generate sensible summary statistics and plots.
2. Modify the prior distribution for `sigma` so that it is Uniform on $(0, 100)$. Rerun and compare results.

3. Modify the prior distribution of `delta` so that given `sigma`, `delta` has a Gamma distribution with mean σ and shape parameter `alpha`, where `alpha` is fixed and defined in the `scalars.dat` file. Hint: YADAS contains a class called `GammaMeanAlpha`.

4. Modify the algorithm so that `sigma` and `delta` are updated using Gaussian proposals on the log scale. Do this by changing their definitions from `new MCMCParameter` to `new MultiplicativeMCMCParameter`. Turn on the automatic step size tuner for these two parameters.

5. Change the output so that only every 10th iteration is sent to a file.

6. Experiment with other examples in the `Examples.zzz` file and discussed on the YADAS web site.

7 Example 2: N-component series system

Your first impression of this example may be that it is a simple problem made hard, as indeed it and many other things in YADAS are. However, this complexity serves to enable many generalizations of this simple problem. Introducing this example serves a couple of purposes: first, it introduces the reliability package of YADAS, and second, it opens discussion of some features we've used in our reliability models that we need to discuss further.

We have an $N-$component series system. We will use the subscripts 1 through N to refer to these components, and the subscript 0 to refer to the full system. Data consist of component tests with $(x_1, \ldots, x_N)$ successes in $(n_1, \ldots, n_N)$ trials, and system tests with x_0 successes in n_0 trials. The unknown parameters are $(p_1, \ldots, p_N)$ (the success probabilities for the components), and we are particularly interested in the system success probability $p_0 = \prod_{k=1}^{N} p_k$.

We assume Beta prior distributions for p_1 through p_N. There are several different cases for the parameters of these prior distributions. We can work with two different parameterizations of the Beta distribution: the standard (a, b) parameterization where the density $f(p|a, b) \propto p^{a-1}(1-p)^{b-1}$, and the mean-precision parameterization where $f(p|\tilde{p}, \nu) \propto p^{\nu\tilde{p}-1}(1-p)^{\nu(1-\tilde{p})-1}$, which satisfies $E(p|\tilde{p}, \nu) = \tilde{p}$ and $\mathrm{Var}(p|\tilde{p}, \nu) = \frac{\tilde{p}(1-\tilde{p})}{\nu+1}$.

- Of course, the prior parameters can be fixed, either at some value such as $a = b = 1 (\tilde{p} = 0.5, \nu = 2)$, $a = b = 0.5 (\tilde{p} = 0.5, \nu = 1)$, or something more informative.

- If multiple components rely on the same prior parameters, we can allow the prior parameters to be random and estimate them. For example, we have used a hierarchical model where the components all have the same prior mean $\tilde{p}$, which is random:

$$p_i \sim \mathrm{Beta}(\nu\tilde{p}, \nu(1-\tilde{p})) \ (i = 1, 2, \ldots, n), \text{ with } \tilde{p} \sim \mathrm{Beta}(a, b). \tag{3}$$

  If I remember correctly, the hierarchical model was proposed by Val Johnson to ensure that if we have only system-level data, the estimate of system reliability does not depend strongly on the number of components in the system. It opens us up to the criticism that we are making the unrealistic assumption that *a priori*, the components are equally reliable. I'm not terribly bothered by this: the belief that the component reliabilities are different may come in part from the tests themselves and thus may not belong in the prior. In a generic system one expects that relatively unreliable components will undergo additional engineering, and this can lead one to expect equal component reliabilities in the absence of other information. Finally, I'll wager a burrito that one doesn't need to assume exactly (3) to get the benefits of the hierarchical model. For example, one can assume

$$E(p_i|\alpha, \beta_i) = \mathrm{logit}^{-1}(\alpha + \beta_i), \text{ with } \alpha \sim N(0, \sigma_\alpha^2) \text{ and } \beta_i \text{ known}. \tag{4}$$

  This preserves the hierarchical model property where we have a parameter α in common across all components, while assuming that *a priori*, some components are more reliable than others. If our customers are unwilling to provide information that we can convert to the β_i, then we can assume (3) with impunity.

We also allow expert opinion about the value of p_0 to be incorporated into the analysis. Conversations with an expert on the system can sometimes in principle be converted to an equivalent number of successes and binomial trials at the system level. This acts like a binomial likelihood for the system reliability $p_0 = p_1 p_2$, and can consequently be read as a beta density term in the posterior.

7.1 Digression

I have come to think that it is quite wrong to call this a "system-level prior." It may have come from the expert's genuine prior distribution for p_0, but it is not our prior for p_0. Our prior for p_0 can only be found by considering our joint prior for (p_1, p_2) and evaluating the induced distribution of $p_1 p_2$. Now, it is legitimate to combine our prior for (p_1, p_2) with all of our pieces of expert opinion and refer to this as some sort of prior. My opinion, which as far as I know no one agrees with, is that the process of interviewing experts to elicit their opinions can only be interpreted as an experiment generating data with likelihood function equal to the term that as a result you include in the posterior density. For example, if an expert tells us that

$$p_0 \sim \text{Beta}(a+1, b+1),$$

we must interpret this as an experiment that generated data (a, b) with likelihood function

$$\frac{C(p_0)\Gamma(a+b+2)}{\Gamma(a+1)\Gamma(b+1)} p_0^a (1-p_0)^b, \tag{5}$$

where $C(p_0)$ is the constant that leads this expression to integrate to one when we integrate out a and b, if it is even integrable. (In practice, we use $C(p_0) = 1$ instead.) I admit that this opinion of mine is completely ignorant of existing work in expert elicitation, and this should offend people, but one can't read everything. Also, this interpretation of expert opinion implies that classical statisticians can use expert opinion almost as effectively as Bayesians, and that will make no one happy. Also, this is very far afield from learning YADAS, but the handling of expert opinion in system reliability is an important issue that we need to discuss in some forum.

7.2 Return from digression

When expert opinion on the system reliability is expressed in terms of a Beta(a, b) distribution, we adopt the convention of adding one to the a and b parameters and have referred to this as "treating the expert opinion as data" because of the interpretation of a Beta prior distribution being expressed as an equivalent sample size and number of successes, and the difference between the Beta prior and the binomial likelihood being -1's appearing in the Beta exponents. One reason is that if we don't, we can't guarantee a proper posterior distribution for $(p_1, \ldots, p_N)$.

Finally, we can put a prior distribution on ν parameters associated with expert opinion when the experts have only given us point estimates. We "always" use $\nu \sim \Gamma(5, 1)$. According to my memory, I pulled this out of thin air one day with no particular justification. The motivation for allowing ν to be random at all is that experts can be wrong, and this allows the expert opinion to be be downweighted when it disagrees with the data.

7.3 On to the YADAS, finally.

The discussion of the models was probably confusing. Here is a simplified description of the models we can fit with the `SystemReliabilityExample` class in the reliability package.

- We have a series system with N components. Their reliabilities are $p_1, \ldots, p_N$, and the system reliability is $p_0 = \prod_{i=1}^N p_i$.
- We have binomial data (x_i, n_i) for $i = 0, 1, \ldots, N$, with subscript zero referring to the system.
- The prior distribution for the component reliabilities satisfies $p_i \sim \text{Beta}(\nu\tilde{p}, \nu(1-\tilde{p}))$ $(i = 1, \ldots, N)$.

- The hyperprior distribution for $\tilde{p}$ satisfies $\tilde{p} \sim \text{Beta}(a, b)$.
- We have expert opinion at the subsystem level which we incorporate with a density term that looks like
$$p_0^{\nu_0 \tilde{p}_0} (1 - p_0)^{\nu_0 (1 - \tilde{p}_0)}.$$
ν_0 and $\tilde{p}_0$ are fixed. $\nu_0 = 0$ is an option, in which case we have no such expert opinion.
- ν is fixed (at 5).

To run this example, you need to obtain the `yadasandreliability.jar` file and put it in your CLASSPATH as you learned to do in the previous lesson. In fact, I would rename it `yadas.jar` and just use it instead of the previous `yadas.jar`. Sorry about that: some of the code in the new jar file depends on code in another jar file written elsewhere, and I didn't want to require you to get that. But for the moment you can use this jar file without getting the other code.

The new jar file contains the standard YADAS distribution and also the reliability package.

To run this example, obtain the file `tutorialexamples.zip` and extract the subfolder `twocomponentsystem` with its four input files to your local system, and go to that directory. Execute the command

```
java gov.lanl.yadas.reliability.SystemReliabilityExample 10000 ./
```

Here you are executing a class that is included with the YADAS and reliability package distribution, so you don't need a compilation step. Note the technique for referring to a class in a specific package: it is like a path, with periods in separating levels in the folder hierarchy. The full names of packages are like URLs in reverse: the yadas website is `yadas.lanl.gov`, so the yadas package's full name is `gov.lanl.yadas`. 10000 is the number of MCMC iterations and the last argument tells YADAS to use the current directory to find the input files and send the output.

Now, as with example 1, we will walk through the `SystemReliabilityExample.java` file line-by-line and discuss new concepts. Recall that the source code is available inside the file `yadasandreliability-src.zip` (or `.zzz`).

```
package gov.lanl.yadas.reliability;
import gov.lanl.yadas.*;
import java.util.*;
```

There is an additional line before the import statements declaring that `SystemReliabilityExample` is part of the `reliability` package of YADAS.

```
public class SystemReliabilityExample {

    public static void main (String[] args) {

        int B = 1000;
        String direc = "/Users/tgraves/projects/yadas/reliability/exampledata/";
        String filename = "components.dat";
        String shortfilename = "leaves.dat";
        String priorfilename = "ptilde.dat";
        String nufilename = "nu.dat";

        try {
            if (args.length > 0)
                B = Integer.parseInt(args[0]);
            if (args.length > 1)
                direc = args[1];
            if (args.length > 2)
                filename = args[2];
```

```
        if (args.length > 3)
            shortfilename = args[3];
        if (args.length > 4)
            priorfilename = args[4];
        if (args.length > 5)
            nufilename = args[5];
    }
    catch (NumberFormatException e) {
        System.out.println("Poor argument list!" + e);
    }
```

The start of the code, as in example 1, declares the name of the class, defines default values for command-line inputs, and processes command line inputs.

```
    DataFrame d = new DataFrame (direc + filename);
    DataFrame leafd = new DataFrame (direc + shortfilename);
    DataFrame priord = new DataFrame (direc + priorfilename);
    DataFrame nud = new DataFrame(direc + nufilename);
```

This example requires four data files. We will discuss these in more detail later, but we note here that these same data files, with minimal changes, can be used to analyze an n-component series system for several values of N. The first line of `components.dat` should be the integer $N+1$, and the first line of `leaves.dat` should be N. As I sent them to you, $N = 2$, but you can do analyses for other N by changing these two lines.

- `components.dat` has a horrible name; `nodes.dat` would be better. (At one point I was using the name "components" to refer to any node in the graph for a system, including genuine components, subsystems, and the full system. It seemed like a good idea at the time. Software has a way of immortalizing your bad decisions.) This file defines the system structure, stores the data at all levels, and defines the mapping between nodes and the parameters $p, \tilde{p}$, and ν. It has one row of data for each node in the graph: for the $N-$component series system, that's $N+1$.
- `leaves.dat` contains one row for each leaf node in the graph; one for each thing we would typically call a "component." It gives initial values for the p_i and step sizes.
- `ptilde.dat` features one row for each $\tilde{p}$, and in our case this is two, one for the system and one for the components. It includes only (initial) values for these parameters and step sizes, which can be zero.
- `nu.dat` contains one row for each ν, which is also one for the system and one for the components. It contains (initial) values, step sizes, and parameters for their gamma distributions. In our example both ν's are fixed (have zero step sizes), and $\nu_0 = 0$. It is important that if one of the νs is zero, its prior parameters are such that the gamma density is positive at zero (for example, $a = b = 1$). Otherwise, the parameters will never move from their initial values. (The reason, if you care, is that the unnormalized posterior is always $-\infty$, so the acceptance probability for all moves is `NaN`, so every move is rejected.)

```
MCMCParameter p, ptilde, nu, notleaf;

MCMCParameter[] paramarray = new MCMCParameter[]
{
    p = new LogitMCMCParameter
        (leafd.r("p"), leafd.r("pmss"), direc + "p"),
    ptilde = new LogitMCMCParameter
        (priord.r("ptilde"), priord.r("ptmss"), direc + "ptilde"),
    nu = new MultiplicativeMCMCParameter
        (nud.r("nu"), nud.r("numss"), direc + "nu"),
    notleaf = new MCMCParameter
        (d.r("notleaf"), d.r(0), direc + "notleaf"),
};
```

The three parameters you expected are all defined. Note that since p and $\tilde{p}$ are probabilities, we have defined them to be `LogitMCMCParameter`s so that Metropolis-Hastings proposals are Gaussian random walk proposals on the logit scale. Similarly, ν is defined to be a `MultiplicativeMCMCParameter` so that it is updated on the log scale. The surprise is the parameter `notleaf`. This thing contains indicators of whether an element of p corresponds to a leaf in the graph. (Ideally YADAS would figure this out on its own, but it doesn't.) We are not seeking posterior samples of this, but we need to define it as a parameter because of a quirk of the YADAS class `FunctionalArgument`, which will be defined below.

```
MCMCBond binomialdata, p_prior, nu_prior;

ArrayList bondlist = new ArrayList ();

BasicSystemProbArgument bspa = new BasicSystemProbArgument
  ( new IdentityComponentProbs (0, d.length() ),
    new BasicSystemConverter
    (d.i("parents"), d.i("gate"), d.i("pexpand")));
```

Usually not much goes on at this stage except declaring some names for bonds and declaring `bondlist`. Here we are defining a special object, a `BasicSystemProbArgument`, which is the key to many YADAS system reliability analyses. We define it before the definitions of the bonds because it will be used in two different bonds. Its definition consists of two parts: a way to calculate the reliabilities of leaves (components) as a function of unknown parameters, then a way to calculate the reliability of subsystems and the full system as a function of leaf reliabilities.

In this example, an object called an `IdentityComponentProbs` is used to compute the leaf reliabilities: in this case, the unknown parameters are exactly the leaf reliabilities, so the object needs only to know how to find the appropriate parameter (parameter 0) and how many nodes are in the system. In more complicated examples, unknown parameters may need to be transformed into reliabilities (in the case of covariates, for example), and in these cases we use some other object that implements the `ComponentProbBuilder` interface like `IdentityComponentProbs` does.

This example also uses a simple class to calculate the upper-level node reliabilities, a `ComponentToDataConverter`. This class needs to be told the graph structure of the system so that it can convert component reliabilities to subsystem and system reliabilities. It accepts this information in the form of three (sometimes four) arrays of integers. The first array describes how to draw the graph: if the ith element of this array is equal to j, then component i is part of subsystem j, and if node i represents the whole system, then the ith element is set equal to -1. In our example, the zeroth element is -1 and the remaining elements are equal to 0. The next array describes the series or parallel structure: if this array's ith element is positive, then the ith node represents a series (sub)system, while nonpositive indicates a parallel (sub)system; if node i is a component, the ith element is ignored. The third array is used for mapping the nodes to the unknown parameters: if node i is not a leaf, it should get a negative value here, while the leaves should be numbered from 0 to one less than their number. Finally, a fourth array can be specified here: it defines the order in which node reliabilities should be evaluated, starting with components and working up. (In theory YADAS could do this itself.) By default (i.e. if this fourth array is not specified), the algorithm starts with p_N and works backward through p_0. If the ith element of this array is equal to j, then node j is the ith reliability to be evaluated (confusingly, the ith element in general has nothing to do with the ith node). With this information, YADAS can compute the reliabilities of all nodes in the graph, regardless of the functional form of the leaf reliabilities.

```
bondlist.add ( binomialdata = new BasicMCMCBond
    ( new MCMCParameter[] { p },
      new ArgumentMaker[]
        { new ConstantArgument (d.r("x")),
          new ConstantArgument (d.r("n")),
          bspa },
      new Binomial () ));
```

We have binomial data throughout the graph, specified in the x and n columns of the input file `components.dat`. This is our first introduction to `Binomial`, but it is probably self-explanatory. The interesting thing here is that the special argument `BasicSystemProbArgument` has been used to calculate the reliabilities of all the nodes in the graph.

```
bondlist.add ( p_prior = new BasicMCMCBond
    ( new MCMCParameter[] { p, ptilde, nu, notleaf },
      new ArgumentMaker[]
          { bspa,
            new FunctionalArgument
                (d.length(), 4, new int[] {1, 2},
                 new int[][] { d.i("priorexpand"),
                               d.i("nuexpand") },
                 new Function ()
                     { public double f(double[] args)
                         { return args[1] * args[2] + args[3];
                     }}),
            new FunctionalArgument
                (d.length(), 4, new int[] {1, 2},
                 new int[][] { d.i("priorexpand"),
                               d.i("nuexpand") },
                 new Function ()
                     { public double f(double[] args)
                         { return (1-args[1])* args[2] + args[3];
                     }}),
          },
      new Beta() ));
```

This is a piece of code which is fairly likely to cure you of wanting to learn YADAS. This term specifies prior distributions for the leaf reliabilities and expert opinion for the other nodes, in the form of

$$p_i \sim \text{Beta}(\nu_{g_1(i)}\tilde{p}_{g_2(i)} + I_i, \nu_{g_1(i)}(1 - \tilde{p}_{g_2(i)}) + I_i), \tag{6}$$

for all nodes i in the graph, where g_1 and g_2 are grouping functions and I_i is 0 if node i is a leaf, and 1 otherwise. The `FunctionalArgument` is actually a very useful device for adding versatility to YADAS, but its interface is not intuitive (it will not be reproduced here as it is given on the YADAS website). Most of the elements in the definition of a functional argument are for the purpose of using the unknown parameters to build a rectangular array, to whose rows can be applied an arbitrary function. The columns `priorexpand` and `nuexpand` in `components.dat` define g_2 and g_1 in (6) in the same way as the group means were assigned to data points in the previous example. Here is some documentation of what these `FunctionalArgument`s do, organized by the arguments inside `new FunctionalArgument`:

- `d.length()` is the number of rows in the matrix we are building. Here, this number is $N + 1$, the number of rows of data in `components.dat`. We are trying to build a function that will be applied once for each node in the graph.
- `4` is the number of parameters (and hence the number of columns in the matrix we are building). Actually we are only going to be using $\tilde{p}, \nu$, and `notleaf` here, but p is also part of this `BasicMCMCBond`, so we need to pretend to use it as well.
- `new int[]` $\{1, 2\}$ indicates which of the parameters need to have subscripting (grouping) variables applied to them. We need to map $\tilde{p}$ and ν to the nodes in the graph, and that is the purpose of the columns `priorexpand` and `nuexpand` in `components.dat`.
- `new int[][] {d.i(''priorexpand''), d.i(''nuexpand'')}`

  is a two-dimensional array in which we list the subscripting variables that we announced our intention of using in the last item in this list. There should be one array for each integer in the previous item.

- The last part of the definition of a `FunctionalArgument` is the `Function` itself. The first function, in somewhat more friendly notation, is $f(p, \tilde{p}, \nu, I) = \tilde{p}\nu + I$, and the second function is $f(p, \tilde{p}, \nu, I) = (1 - \tilde{p})\nu + I$. You should recognize these as the parameters of the Beta distributions in the prior for the leaf reliabilities and the system-level expert opinion. It probably isn't useful to try to get you to understand all the elements of the syntax of this function now, but perhaps it is at least realistic that you could attempt another function in the place of one of these.

Functional arguments can only operate on parameters (not on an array of constants, for example), explaining why we had to define `notleaf` to be an `MCMCParameter` earlier.

Another important note is that since the `BasicSystemProbArgument` was used in two different bonds, and part of its definition was that its `IdentityComponentProbs` was pointing at the 0th parameter, it was necessary that the parameter p occupy the 0th position in the array of parameters in both the `binomialdata` bond and the p prior bond.

```
bondlist.add (pt_prior = new BasicMCMCBond
    ( new MCMCParameter[] { ptilde },
      new ArgumentMaker[]
          { new IdentityArgument (0),
            new ConstantArgument (priord.r("apt")),
            new ConstantArgument (priord.r("bpt")) },
      new Beta () ));
bondlist.add (nu_prior = new BasicMCMCBond
    ( new MCMCParameter[] { nu },
      new ArgumentMaker[]
          { new IdentityArgument (0),
            new ConstantArgument (nud.r("anu")),
            new ConstantArgument (nud.r("bnu")) },
      new Gamma () ));
```

However many $\tilde{p}$s and νs there are, they are given beta and gamma prior distributions with prior parameters given in the two corresponding input files.

```
p.nf.setMaximumFractionDigits(6);
ptilde.nf.setMaximumFractionDigits(6);
```

The default in YADAS is to output only three decimal places of an MCMC iteration of an unknown parameters. This may not be adequate for your example. Since p and $\tilde{p}$ are probabilities that may be close to one, we increase the number of outputted digits to six.

```
MCMCUpdate[] updatearray = new MCMCUpdate[]
{
    p, ptilde, nu,
};
```

The MCMC algorithm consists of (logit-scale) updates of p and of those elements of $\tilde{p}$ that are not fixed, and (log-scale) updates of those elements of ν that are not fixed. In the example data files, all the νs are fixed (i.e. have step sizes of zero), so that update step does nothing. As always, these update steps can be made tunable by converting, for example, `p` to `new UpdateTuner(p)`.

The remainder of the code is identical to that from the first example.

Here are generalizations to this analysis that you can perform with `SystemReliabilityExample` by altering only the input files.

- The system can have many levels, and can be built with system and parallel pieces.
- Data can be present at subsystem levels as well.

- Different prior distributions can be used for the p_i: they can be informative beta distributions, or the components can be divided up into more groups, each of which has its own $\tilde{p}$ and possibly its own ν, or all groups can share the same ν.
- $\tilde{p}$ can be fixed. If there is more than one $\tilde{p}$, any subset of them can be fixed.
- Expert opinion can be included at subsystem levels. The parameters of these expert opinions terms can be shared across nodes and potentially estimated with prior distributions.
- ν (or several ν's) can potentially be given prior distributions and estimated.